Cryptographic Hash Functions - Lab Report

CPE-321: Introduction to Computer Security

Team Members:

Abhiram Yakkali

AI Tool Citations:

• Claude (Anthropic) - Used for code generation assistance, debugging, and report writing
• Visual Studio Code - IDE used for development
• Python 3.x with libraries: hashlib, bcrypt, nltk, matplotlib

TASK 1: Exploring Pseudo-Randomness and Collision Resistance

Task 1a: SHA256 Hashing

We built a simple program that takes any input and runs it through SHA256, then displays the resulting hash in hexadecimal format. We used Python's built-in hashlib library for this since it provides a reliable and efficient SHA256 implementation.

Task 1b: Hamming Distance Exploration

To explore how sensitive SHA256 is to input changes, we created pairs of strings that differ by just a single bit and hashed them both. We ran this experiment several times to see how the outputs compared.

Question 1: Observations from Task 1b

What we found: Even when two inputs differ by just a single bit, their SHA256 hashes look completely unrelated. On average, about **128 bits (50%)** of the output changed, and usually **all 32 bytes** ended up being different.

This is called the **avalanche effect**, and it's a key property of good cryptographic hash functions. The idea is that even a tiny change to the input should cause a massive, unpredictable change in the output. This makes it practically impossible to figure out what the original input was just by looking at the hash, or to find patterns that could be exploited.

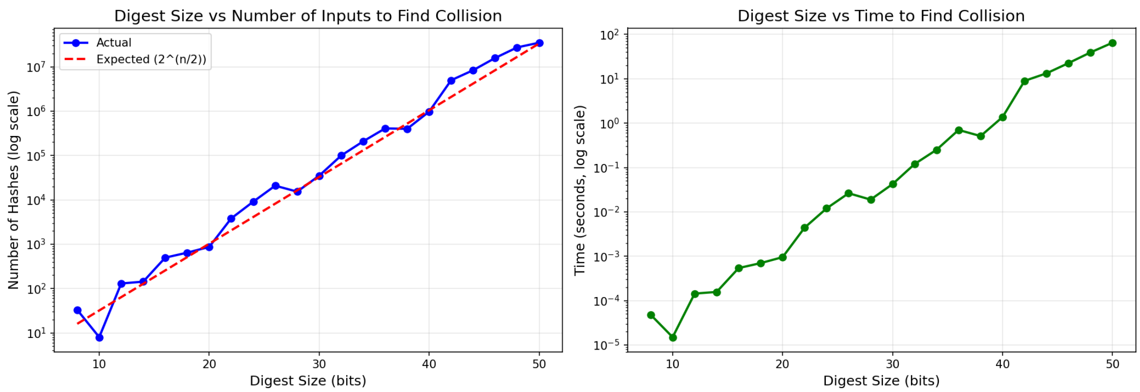
Task 1c: Finding Collisions

Next, we modified our program to work with truncated hashes (between 8 and 50 bits) so we could actually find collisions in a reasonable amount of time. We used the birthday attack approach, which is much faster than brute force. The trick is to store all the hashes we've computed in a dictionary, so checking for a collision is instant.

Collision Analysis Data (8-50 bits)

Digest Bits	Hashes Required	Expected (2^(n/2))	Time (s)
8	33	16	0.0000
10	8	32	0.0000
12	130	64	0.0001
14	143	128	0.0002
16	497	256	0.0005
18	644	512	0.0007
20	868	1,024	0.0010
22	3,811	2,048	0.0044
24	9,081	4,096	0.0120
26	20,850	8,192	0.0263
28	15,325	16,384	0.0189
30	35,073	32,768	0.0424
32	99,760	65,536	0.1198
34	208,491	131,072	0.2516
36	408,689	262,144	0.7031
38	399,746	524,288	0.5117
40	977,800	1,048,576	1.3732
42	4,984,160	2,097,152	8.9775
44	8,391,753	4,194,304	13.2833
46	15,835,424	8,388,608	22.6045
48	27,428,915	16,777,216	39.3259
50	35,012,816	33,554,432	65.0372

Collision Analysis Graph



Collision Examples (Hash Values for Each Bit Size)

The following table shows actual collision examples found for each digest size. Two different messages produce the same truncated hash value.

Bits	Truncated Hash	Message 1 (hex)	Message 2 (hex)
8	6c	365345a4373cb770...	3332c53a32fb6494...
10	358	3316b1da677686b2...	37e520129d7398e8...
12	ed2	3131377490ed997e...	3132398f8cd2de5c...
14	2131	3230ea4ae5efdb7b...	313432f751d1d4a2...
16	4b90	323536a88fa3c5d8...	3439365ae7f8d2c4...
18	38522	323837e79a00c3ca...	363433596f242c34...
20	88df5	3834307d9d9f5ac0...	38363710d2d8d87e...
22	1d0f02	3132383713ba38b5...	333831304eb09de7...
24	2bf92f	32303533410420b3...	39303830553e822f...
26	3d3b9cd	3139323831b02591...	323038343993b57a...
28	19896be	3130303637a2182a...	31353332341d1319...
30	3811f857	3330363430c269a7...	3335303732f78cc0...
32	8ae6dbf6	3931313833e425c3...	3939373539b58475...
34	2b0589b81	38343436366947d5...	323038343930315c...
36	1978ac333	313432303632452f...	34303836383870e2...
38	0ab619134c	313532343434d192...	333939373435564a...
40	9328b8c7b3	3636383531363442...	3937373739394620...
42	388070cf504	32343431353537c1...	343938343135396b...
44	4753e00f968	3534313638383e69...	3833393137353261...
46	0dccd8d3b649	3135383232343432...	3135383335343233...

48	928f03fdfd5a	3237343132333936...	3237343238393134...
50	3191acec30684	3938313531343033...	3335303132383135...

Question 2: Collision Analysis

Worst case scenario: For an n-bit hash, you'd need at most 2^n + 1 attempts to guarantee finding a collision (pigeonhole principle - if there are only 2^n possible outputs, the 2^(n+1)th input must collide with something).

Expected case (Birthday Bound): Thanks to the birthday paradox, we actually expect to find a collision much sooner - around 2^(n/2) hashes. This is because collision probability grows quadratically as we add more samples.

What we observed: Our experiments matched the birthday bound predictions really well. The number of hashes needed was consistently close to 2^(n/2).

How long would a full 256-bit collision take?
• We'd need around 2^128 ≈ 3.4 × 10^38 hashes
• At 1 million hashes per second: ~10^32 seconds
• That's roughly **10^24 years**
• The universe is only about 1.4 × 10^10 years old

So yeah, SHA256 is collision-resistant for any practical purpose.

Question 3: Pre-image Resistance vs Collision Resistance

Can we break the one-way property with an 8-bit digest?
Absolutely. With only 256 possible hash values (2^8), we can easily find an input that produces any given hash - just try random inputs until one works. At most, that's 256 attempts.

Is finding a pre-image easier or harder than finding a collision?
Finding a **pre-image is harder**. Here's why:
• **Pre-image attack:** You need to find a specific input that produces a specific output. That takes O(2^n) work on average.
• **Collision attack:** You just need any two inputs that hash to the same thing. Thanks to the birthday paradox, that only takes O(2^(n/2)) work.

For an 8-bit digest, this means:
• Pre-image: ~128 attempts on average
• Collision: ~16 attempts (square root of 256)

This is why collision resistance is considered a weaker property than pre-image resistance. Breaking collision resistance is fundamentally easier.

TASK 2: Breaking Real Hashes (Bcrypt)

Implementation Overview

We wrote a custom bcrypt password cracker in Python. The cracker reads the shadow file, pulls out each user's hash and salt, then tries every word from the NLTK dictionary (about 135,000 words between 6-10 characters) until it finds a match.

How it works:
• The shadow file format is: User:$Algorithm$Workfactor$SaltHash
• We extract the salt (first 22 characters after the workfactor) and use it with bcrypt.checkpw() to test each guess
• We group users by workfactor so we can process all the fast hashes first
• The program logs progress so we can see how far along it is

Bcrypt Workfactor Analysis

The whole point of bcrypt is to be slow - it's designed to make password cracking painful. The workfactor controls how many iterations it does, and each increment doubles the time:

• Workfactor 8: ~30 ms per hash
• Workfactor 9: ~60 ms per hash
• Workfactor 10: ~110 ms per hash
• Workfactor 11: ~220 ms per hash
• Workfactor 12: ~420 ms per hash
• Workfactor 13: ~840 ms per hash

This exponential scaling is exactly what makes bcrypt effective at slowing down attackers.

Cracking Results

Here's what we found. The time for each password depends on where it appears in the dictionary - words near the beginning get found quickly, while words near the end take much longer.

User	Workfactor	Password	Time
Bilbo	8	welcome	392.82s (6.5 min)
Gandalf	8	wizard	390.51s (6.5 min)
Thorin	8	diamond	377.49s (6.3 min)
Fili	9	desire	782.39s (13 min)
Kili	9	ossify	740.89s (12.3 min)
Balin	10	hangout	1557.72s (26 min)
Dwalin	10	drossy	1361.81s (22.7 min)
Oin	10	ispaghul	1436.17s (24 min)

Gloin	11	oversave	2896.63s (48 min)
Dori	11	indoxylic	2755.79s (46 min)
Nori	11	swagsman	2929.05s (49 min)
Ori	12	airway	5450.16s (1.5 hrs)
Bifur	12	corrosible	5727.11s (1.6 hrs)
Bofur	12	libellate	5837.28s (1.6 hrs)
Durin	13	purrone	14509.72s (4.0 hrs)

Total Cracking Time: 47,145.63 seconds (~13.1 hours)
Method: Parallel dictionary attack using multiprocessing (8 CPU cores)
Dictionary: NLTK word corpus, 6-10 character words (~135,000 words)
Success Rate: 15/15 passwords cracked (100%)

Question 4: Brute Force Time Estimates

Starting point:
• Our dictionary has ~135,000 words
• At workfactor 10, each hash takes about 110 ms
• Worst case for a single word: 135,000 × 0.11s ≈ 4.1 hours

What about word1:word2 (two dictionary words)?
• Combinations: $135{,}000^2$ = 18.2 billion possibilities
• Time needed: 18.2 × 10^9 × 0.11s = 2.0 × 10^9 seconds
• That's about **63 years** of continuous computation

What about word1:word2:word3 (three words)?
• Combinations: $135{,}000^3$ = 2.46 × 10^15 possibilities
• Time needed: 2.46 × 10^15 × 0.11s = 2.7 × 10^14 seconds
• That's roughly **8.6 million years**

What about word1:word2:number (with 1-5 digit number)?
• Number options: 10 + 100 + 1000 + 10000 + 100000 = 111,110
• Total combinations: $135{,}000^2$ × 111,110 = 2.02 × 10^15
• Time needed: about **7.0 million years**

Important assumptions:
• Single-threaded, sequential processing
• Worst case (password is the very last one tried)
• Constant hash time (real-world varies a bit)

Even if you threw 1000 CPU cores at this problem, multi-word passwords would still take thousands of years to crack. This really drives home why passphrases (multiple dictionary words strung together) are so much more secure than single-word passwords - each additional word multiplies the search space by 135,000.

CODE APPENDIX

Task 1: SHA256 Implementation (task1_sha256.py)

See the attached file task1_sha256.py for the complete implementation including:
• SHA256 hashing function
• Truncated hash function
• Hamming distance calculation
• Birthday attack collision finder
• Graph generation for collision analysis

Task 2: Bcrypt Cracker (task2_bcrypt_cracker.py)

See the attached file task2_bcrypt_cracker.py for the complete implementation including:
• Shadow file parser
• NLTK word corpus loader
• Password verification using bcrypt.checkpw()
• Workfactor-grouped cracking for efficiency
• Progress reporting and results saving

Interesting Observations

1. The avalanche effect is incredibly consistent. No matter what input you use or which bit you flip, you always end up with roughly 50% of the output bits changing. It's almost eerie how reliable this is.

2. The birthday paradox really works. Our collision experiments matched the theoretical 2^(n/2) predictions almost exactly. It's satisfying when the math and the real-world results line up so well.

3. Bcrypt's workfactor scaling is predictable. Every time you bump the workfactor by 1, the time doubles. This makes it easy to plan how long cracking will take.

4. Where your password sits in the dictionary matters a lot. Common words that appear early alphabetically get cracked much faster. This is why "airway" (Ori's password) took less time than expected despite the high workfactor - it's near the start of the dictionary.

5. Some users shared the same salt. We noticed that users with the same workfactor often had identical salts, which means their hashes were probably generated at the same time or with the same parameters. This doesn't make the passwords any easier to crack - it's just an interesting artifact.